```
*   procedure IN6
*   procedure ABSA
*   procedure ABSIN
*   procedure ABS
    boolean procedure ATYPE
    boolean procedure BTYPE
    integer procedure EXPAND(X)
    procedure DEFINE(L1, L2)
    boolean procedure LOD
    integer procedure LOWER(X)
*   procedure IDENT
    procedure PAD
    procedure CODE (X)
    procedure COD2(X,Y)
    procedure SIXBIT(CONSTANT)
    procedure LDEC(LNO)
    procedure LABEL (LNO)
    integer procedure JMPNEW
    procedure JMP(LNO)
    integer procedure CJMP
    integer procedure NEWLAB
    procedure NEWADR
    procedure FSCHK
*   procedure SID
    procedure DARR
*   procedure DTV
```

```
*  integer procedure DTYPE
*  procedure INSERT
*  boolean procedure CHKFR
*  procedure GET INTEGER
*  procedure SFR
*  procedure DLAB
*  procedure DELETE
   procedure CSTR
   procedure PCALL
   procedure STATEMENT
   procedure EBLOCK
*  procedure DPROC
   procedure SUBSCRIPT
   procedure FETCH
   procedure EXPRESSION
   procedure RHS
   procedure ASSIGNMENT
   procedure IDOUT
   procedure PR1 (X)
   procedure PR2 (X)
   procedure WARN(X)
   procedure FAIL (X)
   procedure ~~CHK~~ CHFAIL(SYM, FNO)
```

```
        procedure MAKREAL (R1, R2)
        boolean procedure APRIME
        boolean procedure AFAC
        boolean procedure ATERM
        boolean procedure SAE
        boolean procedure AE
        procedure RE
        procedure VARIABLE(LOCAL, GLOBAL, DIRECTION)
*       procedure PUTOUT
*       procedure GETOUT
        integer procedure IFCLAUSE
        procedure BPRIM
        procedure BTERM
        procedure SBE2
        procedure SBE
        procedure SDBE (DBTYP)
        procedure DBE( DBTYP)
*       procedure STID (INDEX, SID1, SID2, SADR)
```

notes on full compiler.

TYPE = laatst gebruikte type van expressie of variabele.

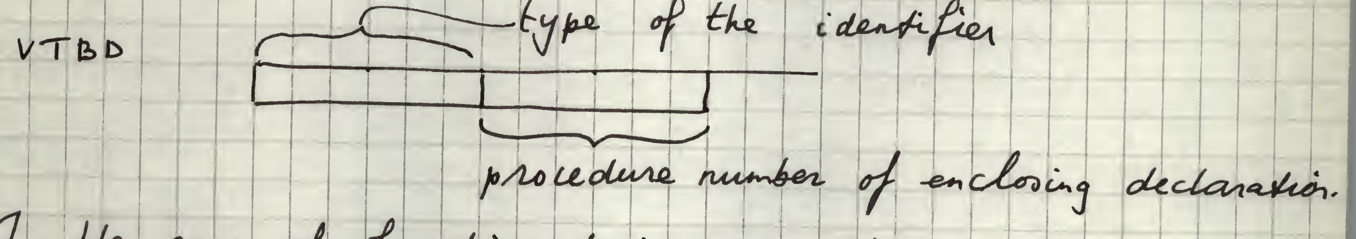
NODEC = totaal aantal identifiers in scope

FOUND = boolean, geeft aan of identifier wel of niet gevonden is.

LDPT = index in FRN1, FRN2, FRN3, FTBD en FLINE voor het opbergen van labels en procedures.

IL1, IL2, IL3, VTBD, VADR are arrays which act as a stack of all declared identifiers.
NODEC points to the top and DBASE to the last identifier in the previous block.

IL1, IL2 and IL3 contain the name of the identifier.



In the case of function designators, the procedure number is that of the actual procedure and not the enclosing one.

The contents of VADR depend on the type of the identifier.

· variables: lower 6 bits contain the stack position of the variable.
formal labels, procedures and strings count as variables for this purpose.
the most significant 6 bits contain the number of subscripts of an actual array.

for labels and procedures VADR contains the L-number of the label assigned to the procedure.

for function designators VADR := 3, the stack position of the result of functions.

FRN1, FRN2, FRN3, FRLN, FTBD and FLINE contain information about actual procedures and labels, wether declared or used before declaration. They are lists whose last element is pointed at by LDPT.

FRN1, FRN2, FRN3 contain the name and FLINE the line number of the use of an undeclared identifier.

FRLN contains the label number assigned to the name. A fresh one is given for each declaration or use, except in the (quite rare) case that the identifier used can be assigned straight away to one that has been declared.

FTBD contains in the least significant 6 bits the type of the identifier.
The most significant 6 bits contain either
a) the block depth of declaration
b) the block depth at which the undefined reference may be equivalenced with a declaration.

Numbering of variables

Results of functions are stored in variable 3, procedure parameters and locally declared variables start at no.4. ~~as an exception,~~

As an exception, main program variables are numbered from 2 upwards, as there is no linking information.

The variable ADDR in the RoGi-Algol full compiler is the above mentioned number.

procedure IN6;

~~conv~~
convert TAB to one single space
if not string skip spaces
skip over linefeed, form feed, @ and return
if return then increment linecount
doll := char = 36 ($)

IN6 leaves last read character stripped to least significant six bits in the integer CHAR

---

procedure ABSA;

reads basic symbols; skips comment.

BS := value of basic symbol
or character if it's not a basic symbol

value of BS: BS := 40 * first char + sec. char.

procedure ABSIN

converts symbols :=, <=, >= to internal representation.

sets boolean TERM to true

if BS = end, ;, else or $.

procedure ABS;

skips comment after end ~~th~~
until first occurence of end, ;, else or $.

sets boolean letter and digit

Note on ~~AS~~ ABSIN en ABS
integer hel1, hel2; boolean hol1, hol2; local hel3;

Bovenstaande variabelen worden gebruikt voor:
— hel1 voor het tijdelijk opbergen van ~~een~~ het eerste character van :=, <= or >=
— hol2 om ~~het [illegible]~~ één character vooruit te kijken
dit is nodig voor ~~z~~ label:....
en :=, ;
immers : kan gevolgd worden door = ⇒ symbool :=
of niet, dan is ~~[illegible]~~ er van een label ~~geweest~~ sprake.

```
procedure   IDENT

—   read first 6 characters of identifier
    and store into integers ID1, ID2 and ID3
         and into integer array ID[1:3]

—   set found to true or false ; if ~~fa~~ true calculate
                                                   DECL

—   if found
    then          TYPE := VTBD[DECL] % 64
     |            ADDR := VADR[DECL]
     |            if array-type then ADDR := lower(ADDR)
    neht
```

procedure SID   Store identifier in list.

globals :   NODEC
              FOUND
              DECL
              DBASE
              TYPE
        procedure FSCHK   Fix space check
              IL1, IL2, IL3 [1:140]
              ID1, ID2, ID3
              VADR, VTBD [1:140]
              PDEPTH , ADDR
              TABLES
        procedure IDOUT   type identifier ID1/ID2/ID3

This procedure stores information:

```
  IL1[NODEC]:= ID1   }
  IL2[NODEC]:= ID2   } name of identifier 6 chars only!
  IL3[NODEC]:= ID3   }
 VADR[NODEC]:= ADDR
 VTBD[NODEC]:= 64*TYPE + PDEPTH
```

ADDR is either the variable number, giving ~~the~~ the position of the variable in the stack relative to the pointer in 1ØØ22, ~~x~~

or it is the label number assigned to a label or procedure.

PDEPTH is the procedure number of the enclosing procedure in which the identifier is declared. The main program is Ø, and procedures are numbered serially as they are encountered, regardless of depth of declaration.
As an exception, ~~the~~ pdepth equals the actual number of a procedure, instead of the number of the enclosing procedure.

TYPE : Ø procedure formal parameter (type not known yet)
1 real ; 2 integer ; 3 boolean
5 real array; 6 integer array ; 7 boolean array
10 procedure
11 real procedure; 12 integer procedure ;
13 boolean procedure.

```
procedure DTV      declare variable list.

begin  integer STYP ;
       STYP := TYPE;        save type of this variable-list
DTV1:  IDENT;               read identifier
       TYPE := STYP;        restore type of this variable-list
       NEWADR;              calculate fresh variable-number
                            ≡ ADDR := NADR ; NADR := NADR + 1
       ~~SID~~
       SID;                 store identifier in list
       if BS = ","
       then  read next symbol ; goto DTV1  neht
             read ~~rest of list~~
                  remainder of ~~list~~ variables.
```

750325-9

<u>integer</u> <u>procedure</u> DTYPE;

Afhankelijk van de waarde van BS
wordt het type bepaald.

```
ϕ  procedure formal parameter (type not known yet)
1  real
2  integer
3  boolean
5  real array
6  integer array
7  boolean array
10 procedure
11 real procedure
12 integer procedure
13 boolean procedure
14 label

4  step
5  array
9  value
```

<u>if</u> basic symbol <u>real</u>, <u>integer</u> or <u>boolean</u>
<u>then</u> verder kijken of next basic symbol <u>array</u> of <u>procedure</u> is. Als dit zo is, type aanpassen door 4, resp 10 op te tellen.

Als geen match gevonden wordt, is het type = ϕ.

<u>procedure</u> P(<u>procedure</u> Q, - - -)

↑

<u>procedure</u> SFR ;

<u>procedure</u> CHKFR (INDEX) ;

<u>procedure</u> INSERT (FORWARD) ;

FRN1, FRN2 and FRN3 contain the name and FLINE the line number of the use of an undeclared identifier.

FRLN contains the label number assigned to the name. A fresh one is given for each declaration or use, except in the (quite rare) case that the identifier used can be assigned straight away to one that has been declared.

FTBD contains in the least significant 6 bits the type of the identifier.
The most significant 6 bits contain either
a) The block depth of declaration
b) the block depth at which the undefined reference may be equivalenced with a declaration.

```
procedure  GET INTEGER ;
if  AE  then    CODE (43)
                       ( fix S1
```

Obviously AE ≡ true for a real valued expression.

```
boolean  procedure  AE ;
if  BS = "if"
then   begin

         end
else  AE := SAE ;
```

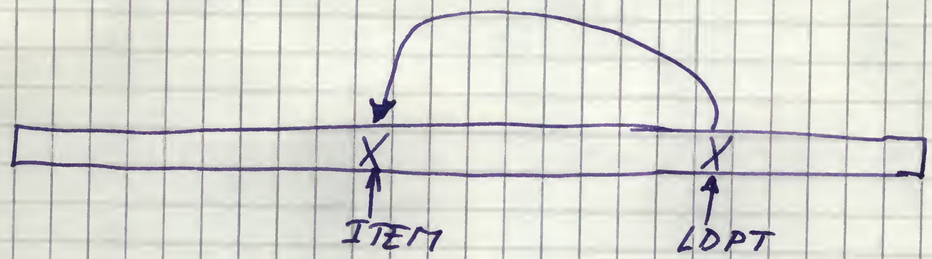
So procedure GET INTEGER compiles
an arbitrary arithmetic expression
and delivers an integer value.

procedure DLAB ;

- LDEC (NLAB) ⇒ PAD ; WRITE(DEV, "L", NLAB, ", ")
- INSERT(∅) ⇒ insert entry in label /procedure list
- delete satisfied forward references
- SID ⇒ store identifier in list
- VADR [NODEC] : = 3 ; ⇒ ADDR := 3

  procedure used as function counts as variable 3.

~~procedure~~ DELETE (ITEM);



ITEM is an index in FRN1, FRN2, FRN3, FRLN, FTBD, FLINE

LDPT points to the last ~~en~~ inserted entry.

CALL OF BUILT IN ROUTINES.

<u>procedure</u> PCALL ;

The first 19 declared variables are standard procedures.

```
if DECL < 10          => I/O procedure.
GET INTEGER     compiles an arbitrary arithmetic
                expression and delivers an integer value.

if SDEC < 7          => 1-5   only one argument
                        6     second argument is string.
else   => decl < 10 and not SDEC < 7

          => so decl = 9, 8, 7

   decl=7   RWRITE ( dev, real expression )
   decl=8   WRITE  ( dev, integer expr.   )
   decl=9   CHOUT  ( dev, integer expr.   )

if SDEC=6

   TEXT ( dev, "string")
or  TEXT ( dev, string-identifier)

if SDEC=6
then begin   - - if BS = " " "
                 then  compile string
                 else  get string-identifier.
```

GETOUT gets outer variable to S1.

CODE (60) => print string whose address is in S1.

so GETOUT yields as result an <u>address</u>!

user-declared procedure calls.

procedure PCALL ;

formal :

```
procedure P(a, b) ; value a; integer a; procedure b ;
begin   }
        |
        b(x)
        }  <- b is used as a formal procedure.
end P ; |
```

formal procedures are local in the enclosing procedure.

In procedure DPROC TYPE := TYPE + 5 if if a parameter is specified a procedure.

call of user declared procedures.

```
procedure PCALL ;

if formal ≠
then    PRAD:= DECL ; STYP:=TYPE-5    neht

else    SFR ; STYP:=TYPE; PRAD:=ADDR  esle

now     PRAD contains procedure-label number
        STYP contains type of procedure.


if  BS # 40        (open parenthesis)
then
          if formal                    get outer variable.
                                       variablenumber= LOWER(VADR[DECL])
          then   GETOUT
                 CODE(58) — enter procedure whose address
                 PAD        is in S1, no parameters.
          neht
          else   CODE(11) — enter procedure with no parameters,
                 LABEL(ADDR)  address in next word.
          esle
goto  ECL2                        leave PCALL
```

user declared procedure call.

```
procedure PCALL;

NPAR: PTYPE := DTYPE;
   if PTYPE ≠ 0
   then  type specification of actual parameter
         may be
              label, procedure, real procedure, integer proc
                     or boolean procedure.
```

750327-75
[illegible]

procedure STATEMENT

springen over procedure - declarations.

na een begin wordt pl8 := false

Dan komt eenmaal JPPL := JMPNEW
en als alle ~~procedure st~~ declarations afgehandeld zijn komt er voor het eerste statement LDEC (JPPL)

Het effect hiervan is als volgt:

```
                              ┌ JUMP. LOCATION IS IN NEXT WORD
                   { CODE (9)
                   { PAD; SIZE := SIZE+1;
JPPL := JMPNEW     { WRITE( DEV,"L", NLAB); SKIP(DEV);
     !             { JPPL := NLAB;
     :             { NLAB := NLAB+1;
                            :
LDEC (JPPL)  \              :
               { PAD;
               { WRITE ( DEV,"L", JPPL,",")
```

```
procedure STATEMENT

    }
    SDBASE := DBASE ;
        }
        |

    DBASE := SDBASE
```

De globale variabele DBASE wordt niet geïnitialiseerd.

Dit is slordig (volgens Rev. Rep. Algol 60) maar het kan hier geen kwaad

Bij binnenkomst in een blok wordt dmv de statements

- SDBASE:=DBASE; DBASE := NODEC

de oude waarde van DBASE gered in SDBASE, nu ~~dan~~ krijgt DBASE een zinvolle waarde in dit blok.

Bij verlaten van het blok wordt dmv de statements

NODEC:=DBASE; DBASE:= SDBASE

de oude situatie weer hersteld.

```
gin procedure P(A); ~~in~~ value A; integer A;
    begin ___
          ___
    end P;
```

procedure STATEMENT;

$\underbrace{\text{JPPL}}_{\text{local}} := \underbrace{\text{JMPNEW}}_{\text{int procedure}}$

end $

$\underbrace{\text{pLS := false ; abm := false ; block := false}}_{\text{lokaal binnen blok" if BS = ~~[illegible]~~85"}}$; $\underbrace{\text{type := dtype}}_{\text{int procedure}}$;

```
TYPE ≠ 0    not BLOK ≡ true
  ⇒ block := true ; depth := depth + 1 ;
     sdbase := dbase ;
  local variable   globale variable.
  dbase := nodec   = total number of identifiers in scope.
  snadr := nadr
~~global~~ local var.   ~~[illegible]~~ global var.
```

75031g-2

__procedure__ DPROC

Save global variables into locals.

```
SLTYP := TYPE;

IDENT      =>   read identifier (name of procedure)
                store into ID
                search declaration list; set found
                calculate TYPE und ADDR
TYPE := SLTYP;

DLAB       =>   Geeft aan de procedure een
                labelnummer en brengt die
                samen met de naam van
                de procedure op.
                Dan doet-ie:
                  PAD; WRITE(DEV,"L", NLAB, ", ")
              Nu is de label NLAB gekoppeld met
              de naam van de procedure en
              ge-identificeerd in de output-code.
```

<u>NOTE</u>: no more than 20 parameters allowed!

<u>procedure</u> DPROC;

compilatie { formele parameters
specification of parameters

```
if BS = 40 then
begin
   I
end parameter specification.
```

— scanning formal parameter list
call of DTV with TYPE = ∅

~~DE~~ DIG[1] through and including DIG[20] set to zero.

— value specification
DIG[DECL − SNODEC] := 1;
every formal parameter specified value gets a 1 on the corresponding place in the array DIG

— type specification

VTBD[DECL] := 64 * SPTYP + PDEPTH

— CODE (number of parameters

— CODE (type of <u>last</u> parameter)
⋮
CODE (type of <u>first</u> parameter)

in reversed order!

<u>formal</u> <u>procedure</u>

— check parameters

<u>if</u> parameter specified as procedure <u>then</u> TYPE := TYPE + 5

<u>if</u> TYPE < 4 (real, int, bool) <u>and</u> DIG[..] ≠ 1 (not spec. value)

<u>or</u> TYPE > 4 (array or proc) <u>and</u> DIG[..] ≠ 0 (spec. value)

<u>then</u> errormessage(41) ≡ simple variable not called by value or other parameter type not called by name.

procedure DPROC

Gegenereerde code :

this number is zero if no parameters specified.

L75 , L76

level of procedure | no of parameters

type param last
⋮
type param first } only if number of parameters # ~~zero~~ ∅

code voor "statement"

if procedure used as function designator COD2(12, 3)
(≡ get local variable 3)

1∅ (leave procedure)

L76 = 27 ;

In this example L75 is the label associated with the procedure. L76 is equated (at the end of the procedure) with a number which represents the ~~space~~ workspace required by the procedure

```
procedure PUTOUT ;
⇒ VARIABLE (4, 37, 26)
             │   │   └ direction
             │   └ global
             └ local.

level := lower( VTBD[ DECL]) ⇒ level := PDEPTH

if level = PDEPTH
then    code (local)  ⇒  code (4)              (a)
else if level = ϕ
    then code ( global)  ⇒  code (37)          (b)
    else cod 2 ( direction, level) ;
            ⇒ cod 2 ( 26, level)               (c).

    code (lower ( VADR[ decl ])) ⇒ code ( ADDR)
                                        = variable-number.
```

so coded is either

a) 4, ADDR   code 4 means: store local variable from S1. followed by variable number.

or b) 37, ADDR   code 37 means: store outer block variable from S1.

or c). 26, level, addr   code 26 means: store to any variable from S1.

<u>procedure</u> GETOUT ; (zie PUTOUT)

⇐ VARIABLE(12, 38, 25)

<u>either</u> a) code (12) ; code (ADDR)

get local variable to S1.
followed by variable number.

<u>or</u> b) code (38) ; code (ADDR)

fetch outer block variable to S1

<u>or</u> c) code (25) ; code (level) ; code (ADDR)

get any variable to S1
followed by level number and variable number.

```
procedure STID(INDEX, SID1, SID2, SADR)
      IL1[INDEX] := SID1 ;
      IL2[INDEX] := SID2 ;
    VADR[INDEX] := SADR
```

store standaard identifier.

```
integer array IL1, IL2, IL3[1:140] , VADR[1:140]
```

bevat de naam van identifier. 2 characters per array.

VADR = variable address
bevat (voor de standaardfuncties) de te genereren code.

```
integer array VTBD[1:140]
```

VTBD contains in the most significant 6 bits the type of the identifier,
and in the lower 6 bits the procedure number of the enclosing declaration.

In the case of function designators, the procedure number is that of the actual procedure and not the enclosing one.

741123 .

Een <program> is òf een <block>
òf een <compound statement>

<block> en <compound statement> kunnen voorafgegaan worden door ~~tot~~ ~~een~~ nul of meer labels

Als we de labels afpellen blijven over:

<unlabelled block> | <unlabelled compound>

Zowel de <unlabelled block> als de <unlabelled compound> beginnen met het symbool "begin".

Dan komen bij de <unlabelled block> een of meer declarations, een puntkomma en vervolgens een compound tail.

Een <unlabelled compound> is van dezelfde structuur, maar daar ontbreken de declarations.

Een <compound tail> tenslotte bestaat uit een of meer statements gevolgd door het symbool "end".

Daarmee is een programma herleidt tot een aantal statements.

```
      procedure print(n); value n; integer n;
begin procedure P(a); value a; integer a;
      TEXT(1, a);
      if n=1 then P("————————————") else
      if n=2 then P("     ————————————") else      ← macro definiëren
      if n=3 then P("     ————————————") else


        @DEF    @NTX : ~~N NO~~ N , text
        if N= ~~NO the~~ @N  then P("@text")  else  ↙)


        @NTX : 1, <————————>
        @NTX : 2, <————————>
        @NTX : 3, <————————>
        @   ,
            ,
            ,
        @NTX : 64, <————————>
```

DECUS EUROPE Ninth Seminar Proceedings 1973

AN EFFICIENT ALGOL-60 SYSTEM FOR THE PDP8

Roger H. Abbott
A.R.C. Unit of Muscle Mechanisms & Insect Physiology
Department of Zoology, University of Oxford
South Parks Road, Oxford OX1 3PS, U.K.

16 december,

Engeland.

— 0944 — 865-56709
Oxford

ABSTRACT

A one-pass compiler translates nearly full Algol-60 into an intermediate language, whose instructions and variable addresses are 6 bits long. The run-time system loads the intermediate language into core memory, and performs the operations specified by its 64 instructions. Execution speed is limited by floating point arithmetic, and is nearly as fast as programs written in machine code. It is about 6 times faster than OS/8 Fortran on a machine with EAE, although compiled programs occupy only one-third of the space. Minimum hardware is an 8K PDP8 with teletype. The system can run under Monitor or OS/8. A 12K machine can use Field 2 for array storage.

INTRODUCTION

The purpose of a compiler is to provide an interface between a program written in a symbolic language and the equivalent binary patterns on which the hardware of the machine can operate. This applied equally to machine code and the so-called high level languages. There are three fairly distinct ways to set about this task.

(a) A compiler can be written to translate the symbolic language into binary or a high level language into symbolic machine code or binary. The machine can then run the compiler output as it stands.

(b) The compiler may translate a high level language into a code which is not machine code, but whose instructions perform the functions which are needed in the high level language concerned. A run-time program then examines these codes, and executes the tasks they specify by means of subroutines. We could include in this class compilers whose output, although machine code, consists mainly of subroutine calls. This latter method is not very efficient, because it takes more instruction bits to specify a hardware subroutine jump and the address of the subroutine than it does to have a code number specifying which out of a list of subroutines should be executed.

(c) The high level language can be stored in the machine as it stands, with no compilation. A run time program interprets it, in a manner similar to (b). This method has the dual advantage that the program is easily modified, and the system can be made conversational. Focal works in this way, as do Basic interpreters. The disadvantage of the method is that programs run relatively slowly. It is not really a competitor to methods (a) and (b), which are used when speed and economy of memory are more important than user interaction with the running of the program.

Since the execution of a high level language requires operations more complex than are provided by the machine instructions of most computers (and certainly the PDP8!), a program translated by method (a) will be longer than one translated into an intermediate code (b), because operations which could be performed by subroutine are done by open code. In the PDP8, floating point arithmetic will limit the execution speed of a well-designed system, because it must be done by software, so method (b) should not be noticeably slower than method (a). A 6 bit instruction allows 64 different codes, which is quite sufficient for running Algol. It also suffices as an address length, since 64 variables in any procedure plus 64 in the main program are adequate. Two 6 bit instructions can be packed into a single PDP8 word. Therefore, method (b) using 6 bit instructions is the best one for the PDP8.

DEC do not offer such a system, the nearest being 4K Fortran which interprets 12 bit codes. It was decided to write an Algol compiler because this is a much more convenient and powerful language than Fortran, and because the PDP8 lacked an Algol-60 compiler.

Design Objectives

As well as being efficient, a high level language system should be convenient to operate. In practice, on a small machine, this means that the translation should involve the minimum number of passes, with the compiler output being as short as possible. The run-time system should also be short, and be designed in such a way that the Algol program can use any peripheral devices that the machine has. It should not be geared to any particular operating system, such as OS/8 or Monitor, but should be capable of running under any such system.

THE OBJECT CODE

It was therefore decided that the Algol should be translated in a single pass into a form which could be loaded directly into memory. Because of the desirability of being able to include machine code statements in the Algol program, the compiler output should be compatible with PAL, so that compiler output and copied machine code could be compiled

together into absolute binary. In the PDP8, it is essential that page boundaries be irrelevant, which means that all label addresses must be 12 bits. (Variable addresses are 6 bits, as already mentioned). This was achieved by having three types of loadable item:

(a) A signed decimal number, which represents two separate 6 bit instructions.

(b) A label address, consisting of the letter L followed by a decimal number.

(c) Floating point literals. These consist of the pseudo-op FLTG, followed by the literal, which is simply copied from the Algol text, followed again by the pseudo-op DECIMAL.

Labels are defined in one of the ways allowed by PAL, either by their occurrence followed by a comma, or by their definition with an equals sign. In the latter case, they are usually equated with a previously declared label. It is a simple matter to have the loader replace these symbolic labels by their binary equivalents. Floating point literals are read into the floating point accumulator by the same routine that reads floating point numbers when the program is running. The loader transfers them to the program area.

THE COMPILER

The most often quoted advantage of Algol over Fortran is that procedures can be called recursively with the evaluation of factorial being used as an example. This is doubly unfortunate, firstly because factorial is most naturally and efficiently evaluated without recursion, and secondly because the main advantage of Algol is that the language is defined recursively. For example, in the condition statement:

if Boolean then S1 else S2;

S2 may be any statement, concluding another conditional:

if Boolean then S1 else if B then S3 else S4;

As a further example, the statement brackets begin.. ...end may be nested, and variables and procedures can be declared after any begin. Evidently, a language which is defined recursively requires a recursively written compiler. Algol provides recursion, and as it is an Algol compiler which we wish to construct, the obvious thing to do is to write the compiler in Algol.

ALGOL -----	COMPILED --------

'IF' B 'THEN' S1;
S2;
	B; JUMP IF FALSE L1;
	S1;↑
	L1: S2;

'IF' B 'THEN' S1
'ELSE' S2; S3;
	B; JUMP IF FALSE L1;
	S1;↑JUMP L2;
	L1: S2;
	L2: S3;

```
'ELSE''IF' BS=366 'THEN''COMMENT' 366 IS 'IF';
'BEGIN''INTEGER' L1,L2;
L1:=IFCLAUSE; 'IF' BS=366 'THEN' WARN(33); STATEMENT;
'IF' BS#212 'THEN' LDEC(L1)
'ELSE''BEGIN' ABS; L2:=JMPNEW; LDEC(L1);
        STATEMENT; LDEC(L2)
        'END'
'END' CONDITIONAL
```

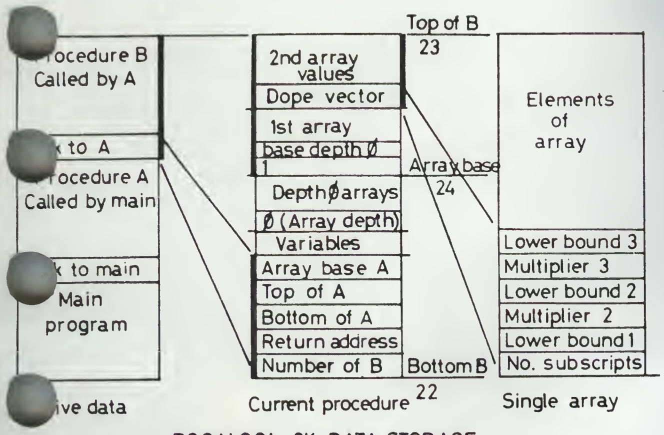
Fig. 1. Section of Algol Compiler

The top part of Fig. 1 shows, on the left, the two possible forms of Algol conditional statment. S2 may be another conditional statement, but S1 may not because the resulting statement is ambiguous. On the right are shown the translated equivalents, with the if, then and else removed. B stands for the code that evaluates the Boolean expression B, and S1, S2 and S3 for the codes that execute the Algol statements of the same names. "Jump if false" is a code whose job it is to examine the result of the Boolean expression, and jump to a label if it is false. Colons signify the definition of a label. Note that the compiled programs are the same up to the arrow. After the arrow, the code depends on whether S1 was terminated by ; or by else. The lower part of Fig. 1 shows the portion of the compiler which deals with conditional statements. It is part of procedure statement, which is called recursively in two places. Because of this recursion the label numbers of the two labels are held in locally declared integers, so that they remain intact through the recursive calls. integer procedure if clause compiles a Boolean expression, checks that the next symbol is then, outputs the conditional jump and returns as its value the label number of the conditional jump. The compiler then checks that the next symbol is not another if (S1 may not be conditional), and if not it compiles S1. Next it checks to see whether S1 was terminated by else (212). If it was not, all it has to do is declare the label L1, but if it was, it must compile a jump to a new label (L2), declare L1, compile S2 and finally declare L2.

The original intention was to write the compiler in full Algol-60, using a full compiler to compile itself. This proved to be impossible because of space problems. Firstly, it is necessary in a full system to check the types of procedure parameters at run time. This check is omitted in the compiler writing Algol system, which saves a great deal of space as the compiler consists mainly of procedure calls (the example in Fig. 1 consists entirely of procedure calls). Secondly, real quantities are not needed in the compiler, and so the compiler operating system does not have routines for dealing with them, leaving more space for identifier tables.

THE RUN-TIME SYSTEM

All run-time programs contain routines for doing arithmetic, evaluating Boolean expressions, entering subroutines and so on. The main feature which distinguishes Algol from Fortran is the way the data is organised, since variables are created as they are declared, and cease to exist when the block in which they are declared is left. In the PDP8 system variable allocation within a procedure is handled by the compiler. In addition recursion must be allowed for. Some text-books, modern ones included, state that this is one of the big difficulties of writing Algol systems, but in reality it is easy. The method is shown in Fig. 2. All that is necessary is to refer to variables by their position in the memory relative to a base pointer. This contains the address of Bottom in Fig. 2. Another pointer marks the next free space at the end of the variables. When a procedure is entered, the base pointer is set to the previous value of the next free space pointer, so that the new procedure has a section of memory all to itself. This arrangement is known as a stack.



ROGALGOL 8K DATA STORAGE

Fig. 2

Within the new procedure's memory are stored the previous values of the pointers, so that the machine can be restored to its previous state when from the procedure is called for. The return ress is also held in this area. The first location in the procedure variable area contains a unique number which identifies the procedure. This is needed when a procedure called at a yet er level refers to variables in the procedure er consideration. It is also used at labels declared in the Algol program, because such labels can be jumped to from procedures active at higher els, in which case the pointers must be reset. compiled program has at every label the identification number of the procedure in which the label is declared. At run time, this number is checked against the identifying number of the procedure el pointed at by the base pointer. If it is erent, procedure levels are removed until the numbers correspond. Jumps into procedures which do not exist in the memory at the time of the jump are prohibited by the compiler.

ays present a special problem because they may appear and disappear within a single procedure and their size is not known until run-time. Arrays are held on a separate stack, which is embedded in the ordinary variable stack. Blocks in which arrays are declared are numbered by depth of declaration. At the beginning of each array level are two words, the first containing the rent declaration depth, and the second the ter to the base of the previous level. When the 12K overlay is in operation, a third word points at the next free space in field 2, where array elements are stored. The array base pointer is ed along with the top and base pointers in the information. Each array starts with a dope vector, which contains all the information necessary to work out the address of an element, given the subscripts. This dope vector is set up at run when the array is declared. In the 8K system, array elements are immediately above the dope vector, but in 12K Algol the last word of the vector contains the address in field 2 where the array begins.

The operating system tape includes the loader, which occupies with its tables the memory which will be used for data storage when the program is running. Currently, the compiler output is loaded into field 0 starting at location 200, but the code is word-wise relocatable, and the system could easily be modified to load and run the code in any part of any memory field.

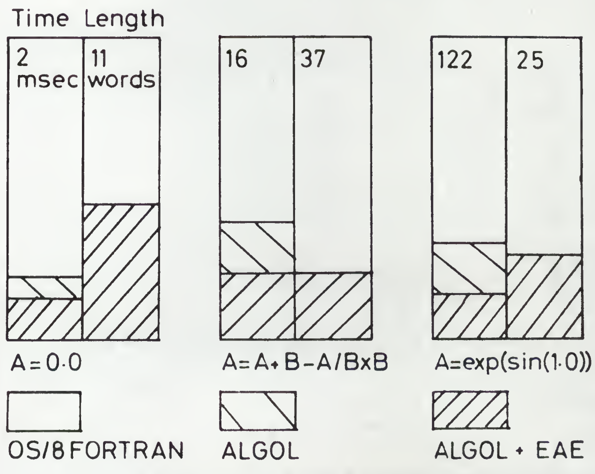
INPUT/OUTPUT

All the built-in input/output procedures have as their first parameter a device number, which must be in the range 0-7. The numbers are logical device numbers, and are used to address a table of input/output machine code routine addresses. Users can assign any device to any number by placing the address of the routine in the table, using an overlay to the run-time system. In the standard system device 0 gives a failure indication in input procedures, but can be used to suppress output by the output procedures. Device 1 is the teletype and device 2 the high-speed reader/punch combination. Device 3 is the systems device, whose routines are written as an overlay to the run-time program, so that various operating systems can be catered for. Currently, Monitor and OS/8 overlays are available. Although the input/output procedures are normally used for just that, the organisation of the run time system allows them to be used for activating any piece of machine code.

SYSTEM PERFORMANCE

Speed

The speed attainable in a program which uses floating point arithmetic is limited by the speed of the floating point software. The statement A:=A+B-A/BxB has been timed in a program written in machine code and in Algol. In a machine which has no EAE Algol is only about 15% slower. If an EAE is available, Algol is about 80% slower, although it is nearly twice as fast as on a machine without EAE. It is believed that this extra time is spent mainly in needless arithmetic stack operations. It is planned to re-write the run time system to avoid these, and when this is done Algol should be nearly as fast as machine code on a machine with EAE.



EXECUTION TIME & CODE LENGTH

Fig. 3

Fig. 3 shows a more detailed comparison between OS/8 Fortran and Algol. In each case, the Algol values are represented as a fraction of the OS/8 Fortran values. Without EAE, Algol is about 3 times as fast, and with EAE about 6 times as fast. Fortran is hardly speeded up at all by use of the EAE, because its speed is not limited by the speed of the arithmetic routines.

Storage requirements

Fig. 3 also shows that the compiled Algol code is only one-third of the length of compiled Fortran code. However, the saving in space is greater, for two reasons. Firstly there is a greater amount of memory available for storing programs. Fig. 4 shows a memory map of an 8K machine, the cross-hatched areas are ones occupied by the system, and

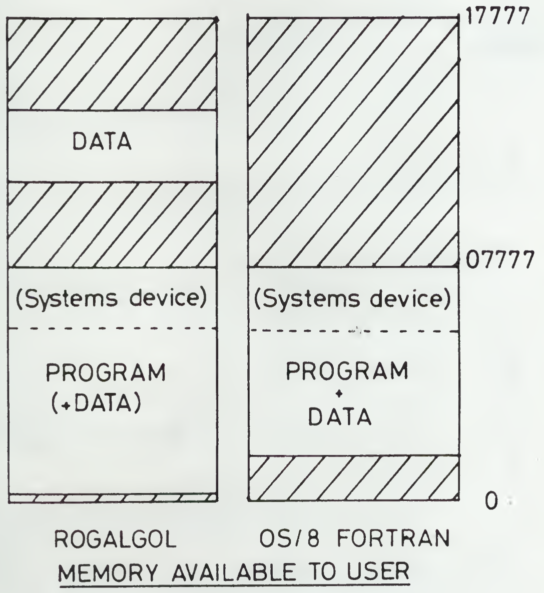


Fig. 4

The hatched areas are occupied by system routines. The items in brackets are optional.

not available to the programmer. The Fortran system is evidently much longer, although it has to be admitted that this is partly due to the greater facilities of the input/output handler.

The second reason is more subtle. When using machine code, we automatically think of writing a program as a series of subroutines, which are often short, because this saves space and makes the logic of the program easier to follow. Fortran is very bad at subroutines, because each one occupies at least one page, and has to be compiled separately. This is sometimes quoted as an advantage of Fortran, and although this may be true in general it is certainly not true of the PDP8 implementation. Algol is efficient in this respect. In the system described here, the minimum length of a compiled procedure is 3 words, compared with Fortran's 128 words.

The Algol compiler is about the same length as the Fortran compiler. The complete Algol run-time routines are about as long as the linking loader program needed by the Fortran system.

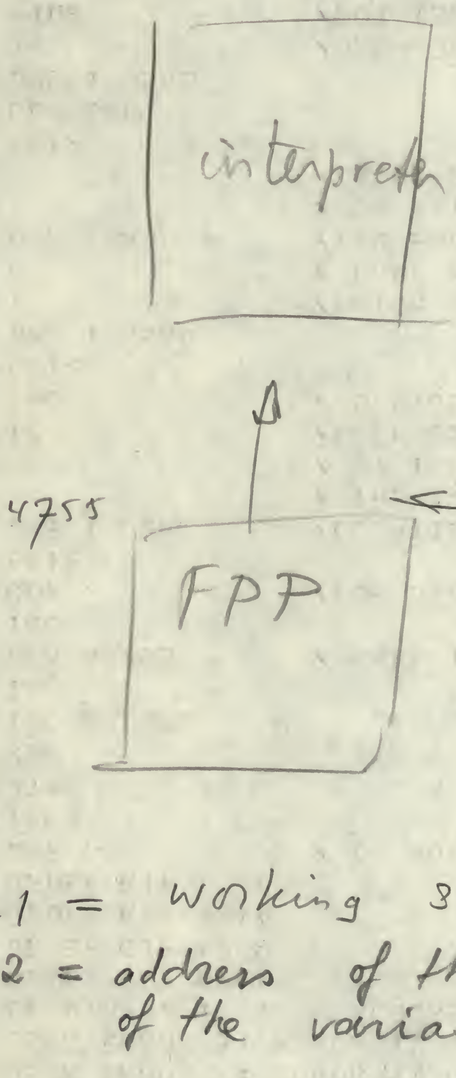
A good starting reference for those wishing to learn more about Algol Compilers is Vol. 3 of Annual Reviews in Automatic Programming.

variables pointer 2∅
locals pointer 22

∅∅176 ?

23 points to next free space.

stackpointer contains initially 4755



stackpointer points to 4755 and goes downwards back to the end of the interpreter (*INT = 2466)

21 = working stack base of current level.
22 = address of the start of the variables of the current procedure.

740122-1

B3PWR 740807-2

```
C        GEWIJZIGD PROGRAMMA

C        P.DAMMAN;B3PWR: A FORTRAN PROGRAM TO EVALUATE
C        SOME LINES OF EQUAL VALUE OF THE FUNCTION: Z=F(X,Y),
C        AND TO WRITE THEM ON THE DISK
         COMMON SCHAR,XMIN,XMAX,YMIN,YMAX,IL,IT,IC,CBE,CBO
         DIMENSION IR(2000),CBE(10),CBO(10),IVERK(2),SCHAR(40)
C        WRITE
C        -----
C        PROGRAM FOR BLOCK TRANSFER IN A FORTRAN PROGRAM
C        USER ERROR 2: ERROR IN USE OF DEV. HANDLER
C        USER ERROR 3: ERROR IN USE OF USR.
C        LENGTH OF IR IS 6 BLOCKS (12 RECORDS)
S        OPDEF CDF 6201
S        OPDEF CIF 6202
S        OPDEF RDF 6214
S        OPDEF RIF 6224
S        JMP GO             / DO NOT RUN INTO HANDLER
SHNDLER,1000
SUSR,    7700
SUSRL,   200
SGO,     TAD HNDLER
S        IAC
S        DCA ARGUS          / MAKE THE ADDRESS OF DEVICE HANDLER
S        IAC
S        CDF                /FOR USR
S        6212
S        JMS I USR          /LOCATION USR MUST NOT BE OFF PAGE BECAUSE
S                           / THE 'JMP I LINK' THAT IS GENERATED
S                           / BY THE SABR ASSEMBLER
S        10                 /LOCK USR IN CORE
S        IAC                / DEVICE NUMBER IN AC
S        6212
S        JMS I USRL
S        1                  /FETCH HANDLER
SARGUS,  0                  / PAGE FOR HANDLER PLUS INDICATION FOR
S        JMP ERROR          /TWO PAGE HANDLER
S                           / ON RETURN ARGUS CONTAINS
S                           /ENTRY POINT OF HANDLER
S        6212
S        CLA IAC
S        JMS I USRL
S        3                  /OPEN OUTPUT FILE
SSTART,  FILE               /POINTER TO FILENAME; ON RETURN
S                           / START CONTAINS THE STARTING BLOCK NUMBER
S                           / OF THE FILE. START# CONTAINS LENGTH
S                           / OF THE FILE
S        0
S        CDF
S        6212
S        JMS I USRL
S        11                 /CALL USROUT IN ORDER TO RESTORE COMMON
C         NOW THE DEVICE HANDLER IS IN CORE ON PLACE HNDLER,
C        A TENTATIVE FILE HAS BEEN OPENED WITH NAME CNTOUR.DA
C        WITH STARTING BLOCK NUMBER ON START
S        JMP COMP
SCLOSE,  RIF
S        TAD FIELD
S        DCA CFL
S        CPAGE 12
S        IAC
SCFL,    0
S        6212
S        JMS I USR2
```

code 4 < 3 words!

Store local variable from SI. followed by variable number

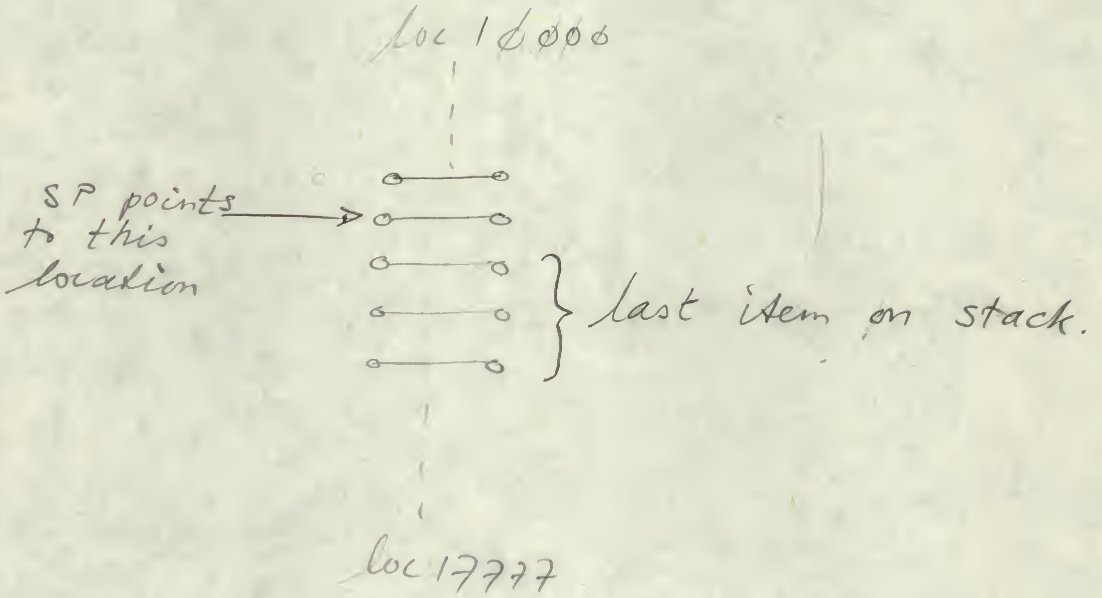
```
2214 PUT

PUT,  NEXT6      /get variable number in AC      ~~JMS I XNEXT6~~ (200) JMS 200
      VADR       ~~JMS I XVADR~~ (325) JMS 325
      TAD 22     /AC contains 3*(varno-1)
      IUNSTAK    ~~JMS I XIUNST~~ (310) JMS 310
      JMP I NEXT+1
```

22 points at start of current level.

enter XIUNST with $AC = (22) + 3*(\text{varno} - 1)$

$\equiv$ address of variable!

SP
SP+1
SP+2



740122-2

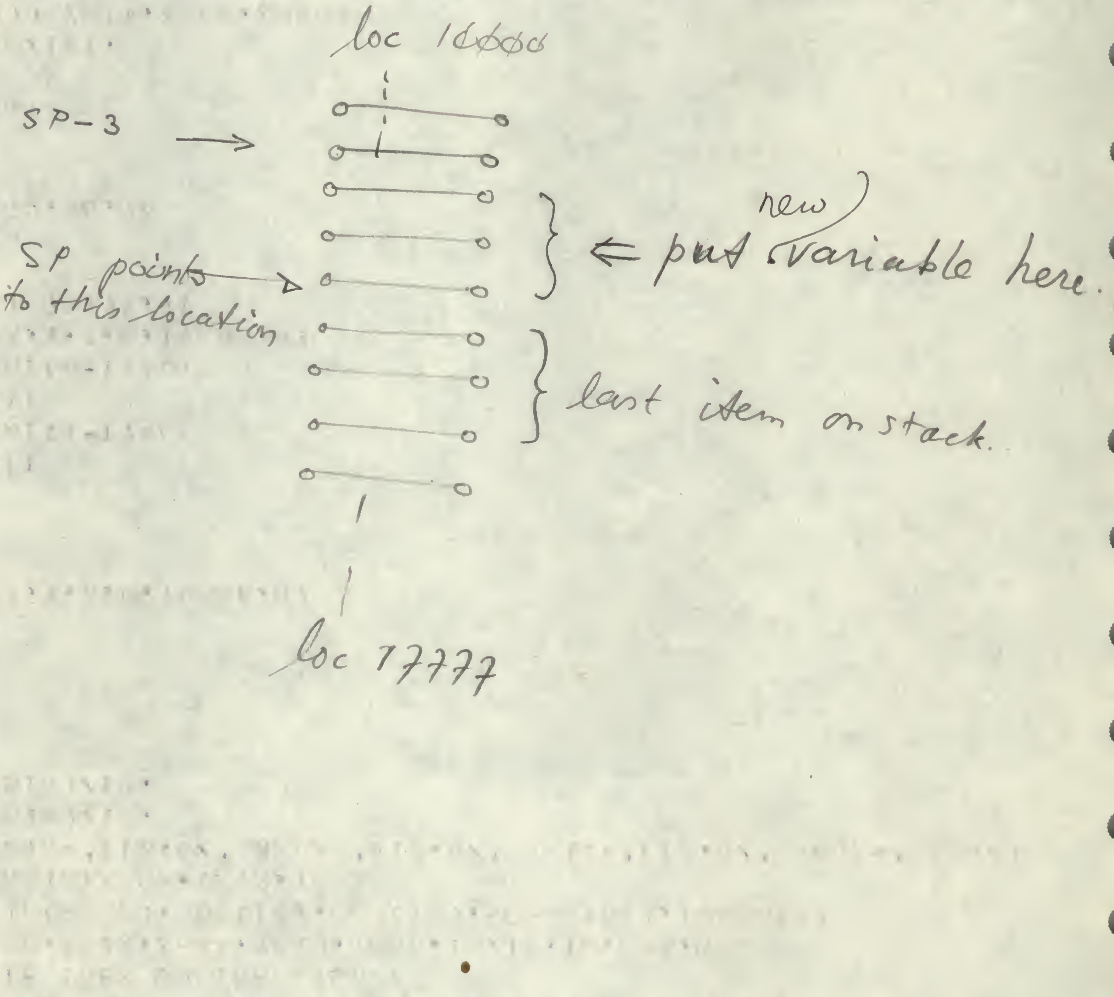
```
      CBO(K)=CBO(K)+WIDTH
98    CONTINUE
110   IC=I
      DO 102 K=1,I
      WRITE (1,103)CBE(K),CBO(K)
102   CONTINUE
103   FORMAT (5H CBE=,F10.4,5H CBO=,F10.4)
      CALL CHAIN ('B3PWR')
      END
```

code $14_8 = 12_{10}$ → 3 words!

get local variable to S1. followed by variable number.

loc. 10114 / 1664 GET

```
GET,    NEXT6
        VADR
        TAD 22
        ISTAK
        JMP NEXT
```



740122-3

A3PWR 740807-1

```
C       P.DAMMAN;A3PWR: A FORTRAN PROGRAM TO EVALUATE
C       SOME LINES OF EQUAL VALUE OF THE FUNCTION: Z=F(X,Y),
C       AND TO WRITE THEM ON THE DISK
        COMMON SCHAR,XMIN,XMAX,YMIN,YMAX,IL,IT,IC,CBE,CBO
        DIMENSION IR(2000),CBE(10),CBO(10),SCHAR(40),IVERK(2)
        READ(1,11)XMIN,XMAX,YMIN,YMAX
11      FORMAT(' XMIN='F10.4/' XMAX='F10.4/' YMIN='F10.4/' YMAX='F10.4)
        DX=(XMAX-XMIN)/10.
        DY=(YMAX-YMIN)/10.
        Z=0.
        X=XMIN
        Y=YMIN
        IVERK(1)=0
        IVERK(2)=1
        CALL FUNC(X,Y,Z,G,IVERK,A)
        ZMIN=Z
        ZMAX=Z
        ZMEAN=0.
        DO 70 I=1,11
        X=XMIN+FLOAT(I-1)*DX
        DO 70 J=1,11
        Y=YMIN+FLOAT(J-1)*DY
        CALL FUNC(X,Y,Z,G,IVERK,A)
        IF (Z-ZMAX) 73,71,71
71      ZMAX=Z
        GOTO 72
73      IF(ZMIN-Z)72,74,74
74      ZMIN=Z
72      CONTINUE
        ZMEAN=ZMEAN+Z
70      CONTINUE
        ZMEAN=ZMEAN/121.
        WRITE (1,77) ZMIN,ZMAX,ZMEAN
77      FORMAT (6H ZMIN=,E10.3,8H   ZMAX=,E10.3,8H  ZMEAN=,E10.3)
75      PMAX=ZMAX-ZMEAN
        PMIN=ZMEAN-ZMIN
        IF(PMIN-PMAX)80,80,90
80      CINT=PMIN/3.
        I=1
        CBO(I)=ZMIN+CINT
82      IF(CBO(I)-ZMAX)83,83,94
83      IF(I-10)85,100,100
85      CBO(I+1)=CBO(I)+CINT
        I=I+1
        GOTO 82
90      CINT=PMAX/3.
        I=1
        CBO(I)=ZMAX-CINT
92      IF(ZMIN-CBO(I))93,93,94
93      IF(I-10)95,100,100
95      CBO(I+1)=CBO(I)-CINT
        I=I+1
        GOTO 92
100     WRITE (1,101)
101     FORMAT ('MORE THAN 10 CONTOUR LINES FOUND')
        I=I+1
94      WIDTH=CINT*0.15
        I=I-1
        DO 98 K=1,I
        CBE(K)=CBO(K)-WIDTH
```

disk (2[illegible], '—")

*

$11_8 = 9_{10}$ JUMP. location is in next word.

607 J

```
J, CLA CMA      / AC := -1  because PC = auto-index reg.
   PARAM        JMS 167.1
   DCA PC
   JMP I NEXT+1
```

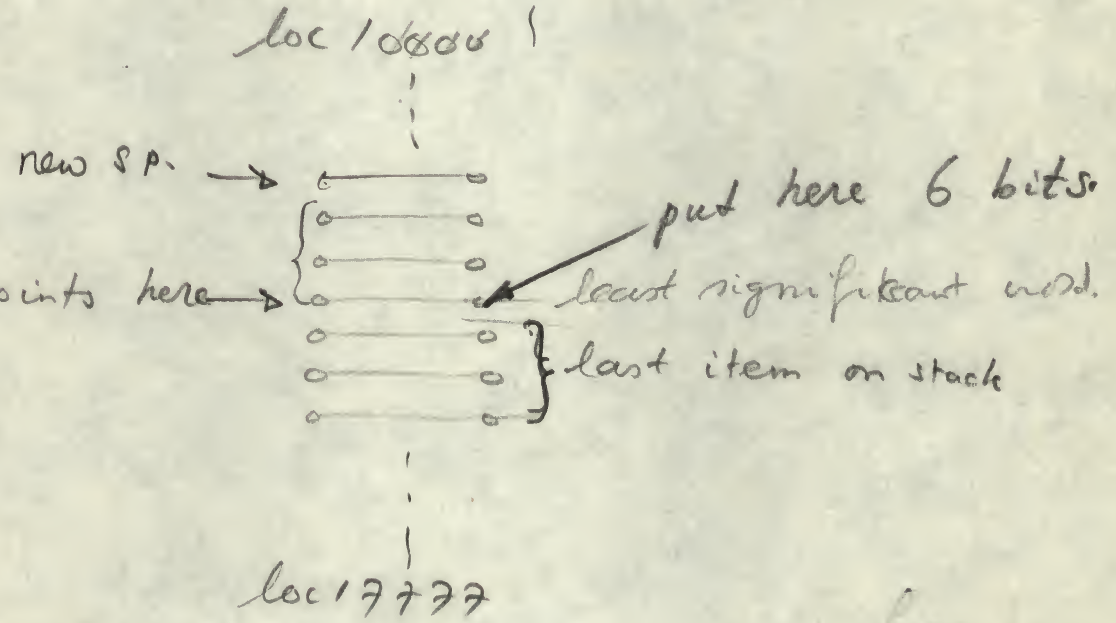
740122-4

```
.R EDIT
*9P\P\DET<9DET

#R

#L
'BEGIN''INTEGER'N,I,J;
'REAL''PROCEDURE'DET(N,A);
  'VALUE'N;'INTEGER'N;'ARRAY'A;
'BEGIN''INTEGER'I,J,K;'REAL'S;
  'IF'N=1'THEN'DET:=A[1,1]
  'ELSE''IF'N=2'THEN'DET:=A[1,1]*A[2,2]-A[1,2]*A[2,1]
  'ELSE'
  'BEGIN'S:=Ø;
    'FOR'K:=1'STEP'1'UNTIL'N'DO'
    'BEGIN''ARRAY'B[1:N-1,1:N-1];
      'FOR'I:=1'STEP'1'UNTIL'N-1'DO'
        'FOR'J:=1'STEP'1'UNTIL'N-1'DO'
         B[I,J]:=A[I+1,'IF'J<K'THEN'J'ELSE'J+1];
      S:=S+('IF'K%2*2=K'THEN'-1'ELSE'1)*A[1,K]*DET(N-1,B)
    'END'FOR K;
    DET:=S
  'END'
'END'DET;
  N:=READ(1);
  SKIP(1);
  WRITE(1,N);
  SKIP(1);
  'BEGIN''ARRAY'A[1:N,1:N];
    'FOR'I:=1'STEP'1'UNTIL'N'DO'
      'FOR'J:=1'STEP'1'UNTIL'N'DO'A[I,J]:=I*N+J↑2;
    RWRITE(1,DET(N,A))
  'END'
'END'$

#E

.R EDIT
*9PBN<9PBN

#R
?
#L

DECIMAL;FIELD Ø;-885;-1153;576
L2Ø
L21,L22
66
322
772
-1407
-145Ø
1792
L23
-1407
-1407
-1406
773
9ØØ
201
L24
L23,772
-1406
-145Ø
1792
```

$52_8 = 42_{10}$ SET 6 BIT CONSTANT

SETH 5176

```
15176/4464 SETH,   NEXT6
15177/5435         JMP I  PNEXT+1

10034  5435  PNEXT, JMP I .+1
10035  0222         PNEX

10222  3432  PNEX,  DCA I  SP
                    JMS  DECSP
               NEX,
```



stackpointer always points to fist free place on the stack.

```
70
1792
L35
L32,L38,-1407
263
-1088
L39
269
1865
L41
L39,1886
L41,-1406
-1407
772
-1407
1378
71
1792
L40
L37,774
775
-1406
779
774
-1407
844
460
553
1564
L42
775
576
L43
L42,775
-1407
832
L43,-1406
773
911
781
-448
L40,780
-448
L35,777
776
-1406
1322
147
776
-1450
1792
L44
-1407
1097
L45
L44,-1407
L45,-1407
776
-1406
773
947
-884
298
85
-1406
```

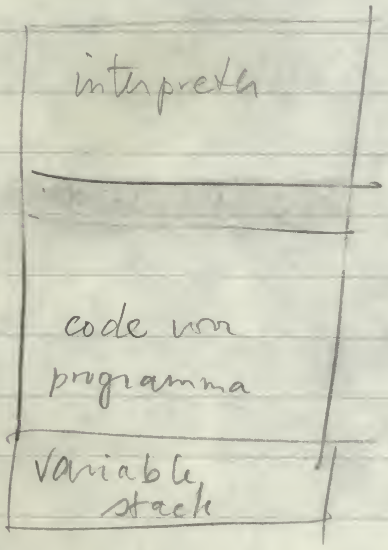
ROG- Algol.

4k Algol genereert binary
beperking : geen user-defined procedures.

ROGER H. Abbot Oxford University dep of Zoology

PDP-8 heeft te beperkte instructieset voor Algol
12 bit woorden
$\Longrightarrow$ 64 typen instructies.

- I/O
- Jump (cond)
- aritmetisch $\lessgtr$ int / float
- procedure calls
- for statement calculata
- push, pop.



implementatie

bootstrapping

full compiler ⟷ integer subset.

eerste gedachte:
1 full compiler geschreven in Algol
2 compileren mbv 4k Grenoble algol of met de hand
3 dit resultaat gebruiken

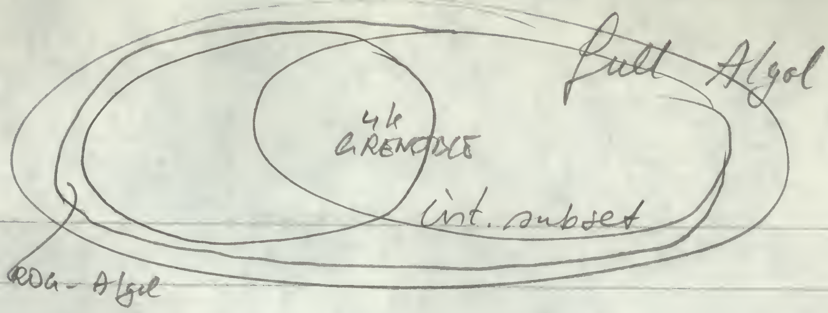
ruimtegebrek

⇒ {integer runtime system
{integer subset compiler

de eerste keer met de hand vertaald naar 6 bit instructie

⇒ werkende integer subset compiler

2



Dan:

full compiler geschreven in integer subset language

laten vertalen door int. subs code + int. runtime system

⇒ 6 bits instructies die moeten worden
geïnterpreteerd door full runtime system.
(o.a. vanwege floating point routines en andere stackopbouw)

Het is mogelijk om machine-code tussen te voegen

trans — leave interpreter

algol — enter interpreter

Special character $\$ = terminator

— conversie basic symbols naar interne representatie

BS := 40 * CHAR1 + CHAR2

A – Z   1 — 26

$$\overset{B}{\overset{|}{}}$$

'BEGIN' := 2×40 + 5 = 85

vgl <u>trans</u> en <u>true</u>

INb : TAB := SPACE ; LINEF, FORMF, @ wordt geskipt
ABSA : lees basic symbols or value of char if not basic symbol
skip comment
ABSIN : :=, <=, >=,
ABS : skip comment after <u>end</u> until ; <u>end</u> <u>else</u> or $ met.

```
procedure STATEMENT

  if letter then begin ident;
                          if colon then label
                    end
else if [ or := then  assignment

else  if  "if"     then   if clause
else  if  "goto"   then  goto statement
else  if  "for"    then  forstatement
else  if  "begin"  then  block or compound
else  if  "trans"  then  machine code statement
```

```
<program> ::= <block> | <compound statement>

<block> ::= <unlabelled block> | <label>: <block>
<compound statement> ::= <unlabelled compound> |
                         <label>: <compound statement>

<unlabelled compound> ::= begin <compound tail>
<unlabelled block> ::= <block head>; <compound tail>

<block head> ::= begin <declaration> | <block head>; <declaration>

<compound tail> ::= <statement> end |
                    <statement>; <compound tail>
```

⇒ programma

```
            begin
                    declaraties
                    statements
            end $
                ↑ ROG Algol.
```

```
<statement> ::=
    <unconditional statement>      assignment statement
                                   goto statement
                                   dummy statement
                                   procedure statement.
    <conditional statement>        if
    <for statement>                for

    <unconditional statement> ::=
        <basic statement>|
        <compound statement>|
        <block>

    <conditional statement> ::=
        <if statement>|
        <if statement> else <statement>|
        <if clause> <for statement>|
        <label> : <conditional statement>

    <for statement> ::=
        <for clause> <statement> |
        <label> : <for statement>
```